Algorithmic Methods of Data Mining
CS-E4600

Programming Project:
Graph Clustering into communities

Alvaro Órgaz Expósito
Student number: 802101
alvaro.orgazexposito@aalto.fi

Adrià Cabeza Sant'Anna
Student number: 784546
adria.cabezasantanna@aalto.fi

December 9, 2019

Contents

1 Introduction

The following report describes the analysis and explanation of diverse methods used for the following graph partitioning task.

Given an undirected graph $G = (V, E)$ and an integer $k > 1$ we want to partition the set of vertices V into k communities $V_1, ..., V_k$, so that $\bigcup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \emptyset$ for all $i \neq j$. We want the communities $V_1, ..., V_k$ to be as much separated from each other as possible. We also want that the communities have roughly equal size. Thus, we will evaluate the goodness of a partition $V_1, ..., V_k$ using the following objective function:

$$\phi\left(V_1, \ldots, V_k\right) = \sum_{i=1}^{k} \frac{\left|E\left(V_i, \bar{V}_i\right)\right|}{|V_i|} \tag{1}$$

where for $S, T \subseteq V$ with $S \cap T = \emptyset$ we define $E(S, T)$ to be the set of edges of G with one endpoint in S and the other endpoint in T, i.e., $E(S, T) = \{(u, v) \in E | u \in S \text{ and } v \in T\}$, and we also define $\bar{V}_i = V \backslash V_i$.

The graphs that will be used are the largest connected component of 5 graphs from the Stanford Network Analysis Project (SNAP):

Graph	**#vertices**	**#edges**	**#clusters**
ca-GrQc	4158	13428	2
Oregon-1	10670	22002	5
soc-Epinions1	75877	405739	10
web-NotreDame	325729	1117563	20
roadNet-CA	1957027	2760388	50

Table 1: Graphs statistics.

2 Literature review

The task of partitioning a graph has grown quite popular since we find ubiquitously in real-world network several sets of densely connected nodes, joined by a small number of edges. Moreover, as it can be seen in Figure [1], this task is not trivial at all as normally no clear partitions are found in the graph, in fact, graph partitioning is an NP-hard problem. Therefore, there are a lot of different approaches to tackle the problem, so in the following section we will introduce some of them in order to get an essential view of the problem and its complexity before implementing our methods.

2.1 The Girvan-Newman method

The method introduced in 2002 by Girvan M. and Newman M. [1] is based on the fact that edges connecting communities should have high **betweenness centrality** which means that for every pair of vertices in a connected graph, there exists at least one shortest path. The betweenness centrality for each vertex is the number of shortest paths that pass through the vertex. Therefore, it removes links in order to decrease betweenness and returns the remaining components of the network as the communities obtained. Its main downside is that it makes heavy demands on computational resources, running in $\mathcal{O}(m^2 n)$ with m edges and n vertices or $\mathcal{O}(n^3)$ on a sparse graph.

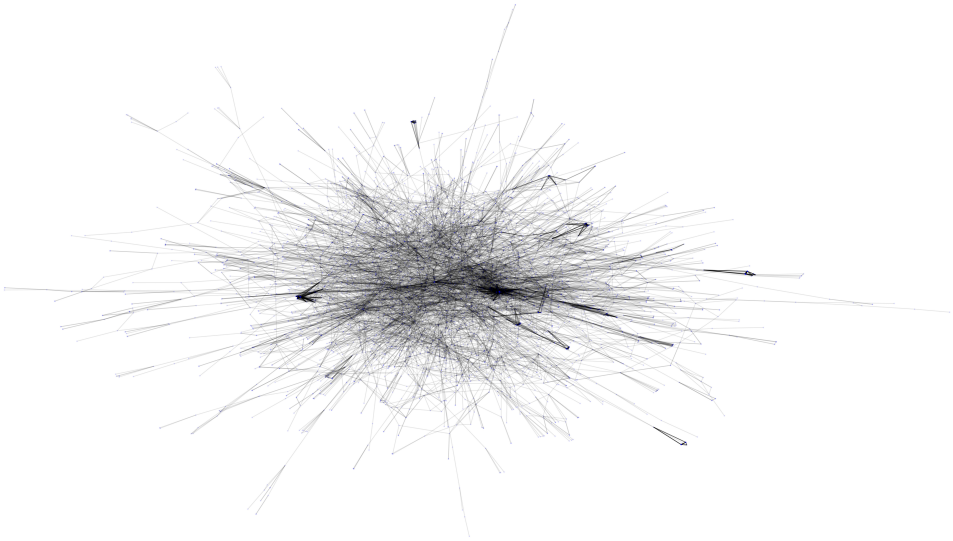

Figure 1: Spring representation of the ca-GrQc graph.

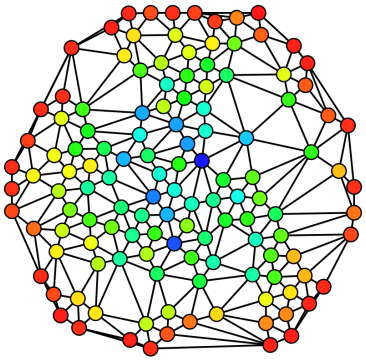

Figure 2: An undirected graph colored based on the betweenness centrality of each vertex from least (red) to greatest (blue). Source: Wikipedia.

Algorithm 1: The Girvan-Newman method.

1. Calculate the betweenness for all edges in the network.
2. Remove the edge with the highest betweenness.
3. Recalculate betweennesses for all edges affected by the removal.
4. Repeat from step 2 until no edges remain.

2.2 CNM method

The Clauset, Newman and Moore (CNM) method [2] is based on increasing **modularity** of a network. The modularity is an evaluation metric for community detection and it tests a given division of a network against the random division. The algorithm measures when a division is a good one, in the sense that there are many edges within communities and only a few between them. It starts with N communities and at every step of the algorithm two communities that contribute maximum positive value to global modularity are merged.

2.3 Spectral methods

Spectral clustering methods use the spectrum of a graph to perform dimensionality reduction before clustering in fewer dimensions. It is better to use this approach than directly using K-means because K-means performs poorly since it can only find spherical clusters. The idea is based on computing the eigenvectors corresponding to the second-smallest eigenvalue of the normalized Laplacian or some eigenvector of some other matrix representing the graph structure [3], [4], [5]. The resulting eigenvector is used as a vertex embedding of the graph to determine the clustering. Its main downside is that computing eigenvalues and eigenvectors for graphs are slow and hence such methods might face scalability issues when applied to massive graphs. Normally, one common way to tackle this problem is to use the more efficient method *Implicitly Restarted Lanczos* where only the k largest or smallest eigenvalues and its eigenvectors are needed.

For example, if a graph G is a collection of k disjoint cliques, the normalized Laplacian is a block-diagonal matrix that has eigenvalue zero with multiplicity k and the corresponding eigenvectors serve as indicator of the membership of each clique: the eigenvector v_i has a different value or a larger magnitude for the vertices that are inside the clique i than the other vertices. Thus, there is an underlying structure that can be seen using the eigenvectors of the Laplacian. Moreover, if we introduce edges between the cliques we will find that $k-1$ of the k eigenvalues that were zero will become slightly larger than zero so it is a robust method.

2.3.1 Recursive Spectral Bi-partitioning

This method uses one of the typical ideas of spectral clustering: computing the eigenvector corresponding to the second-smallest eigenvalue of the normalized Laplacian. The resulting eigenvector works like an embedding of the vertices to split the graph into two clusters: a positive value in the i position of the eigenvector indicates that the vertex i belongs to a cluster C_1 and a negative value that it belongs to a cluster C_2.

In order to do more than two partitions, we can perform a two-classification iteratively, using the spectra of the resulting induced subgraphs. This will yield a divisive hierarchical clustering algorithm that we will call Recursive Spectral Bi-partitioning.

Algorithm 2: Recursive Spectral Bi-partitioning algorithm.

Input: A graph G and the desired clusters k
Pre-processing: Build Laplacian matrix L of the graph
Decomposition:

- Find eigenvectors X and eigenvalues λ of the matrix L
- Sort the eigenvalues: $\lambda_1 < \lambda_1 < \lambda_2 ... < \lambda_n$
- Map vertices to corresponding components of the eigenvector corresponding to λ_2

Grouping:

- Sort components of reduced 1-dimensional vector
- Identify clusters by splitting the sorted vector into two: negative and positive
- Induce two subgraphs using the clusters identified

Recurse with the induced subgraphs until k partitions are reached

---

2.3.2 K-Way Spectral Clustering

Another popular method to tackle the multicluster problem that is more commonly used is the so-called K-way Spectral Clustering introduced by Jianbo Shi and J. Malik [6].

This method instead of using only the eigenvector associated to λ_2 from the Laplacian matrix it uses multiple eigenvectors to build a reduced space where some structural information is conserved. Afterwards, any multidimensional clustering algorithm can be performed such as KMeans.

Algorithm 3: K-way Spectral Clustering algorithm.

Input: A graph G with n nodes and the desired clusters k
Pre-processing: Build Laplacian matrix L of the graph
Decomposition:

- Find eigenvectors X and eigenvalues λ of the matrix L
- Build embedded space from the eigenvectors corresponding to the k smallest eigenvalues

Clustering:

- Apply KMeans to the reduced $n \times k$ space to produce k clusters

3 Implemented algorithmic approaches

Our approach is based on the algorithm [3] mentioned in the previous section. We have started implementing the algorithm on Python3 using **numpy** for basic mathematical operations, **networkx** to deal with the graphs in a sparse way and create their adjacency matrix as well as the normalized Laplacian matrix, **sklearn** for the KMeans algorithm with default KMeans++ centroids initialization as well as Euclidean distance, and **scipy** for the eigendecomposition. Also, we have implemented the target score function to score the clustering output.

Our first implementation was not able to do the eigendecomposition for the largest graphs because we were not using sparse matrices. For the largest graph, roadNet-CA, we had to create an adjacency matrix of $1957027^2 \times 4B$ ($V^2\times$ bytes of an integer) = 14268 GB of memory. However, we noticed that thes graphs are very sparse since the number of edges is highly smaller that V^2, so the first improvement we made was using sparse matrices.

After implementing this improvement, we have added several different configurations and parameters to customize the algorithm and develop deep experiments and find the optimal solutions for all the graphs. This is the list of the principal extensions or added options which can be found in the parameters of the main Python script *graph_clustering.py*:

- Compute the adjacency and Laplacian matrixes manually using sparse functions with the package **scipy** instead of using already implemented functions from **networkx**.
- Compute the unnormalized Laplacian matrix.
- Normalize the eigenvectors by nodes or by eigenvectors axis.
- Invert the eigenvectors multiplying by -1 the Laplacian matrix and selecting the eigenvectors with higher eigenvalues.
- Select only the second eigenvector as an embedding (only 1 dimension by node) for the graph clustering.

- Different clustering methods: KMeans from **scipy** and sklearn, XMeans and Hierarchical Agglomerative Clustering from **pyclustering** and KMeans from **pyclustering** for a more customized version of the algorithm. Note that we use the default Euclidean distance metric and centroids initialization for all the clustering methods except the KMeans method from pyclustering.
- Iterate different random seeds for the selected clustering method and store its best output.
- Customized distance metrics (Minkowski, Chebyshev, Euclidean) and centroids initialization (random and based on KMeans++ algorithm) for the **pyclustering** KMeans option.

Note that there are several ways to compute the Laplacian matrix but we have considered two versions:

- Unormalized: $L_u = D - A$
- Normalized: $L = I - D^{-1/2}AD^{-1/2}$

4 Experimental results

Since we have been competing in the competition and we have implemented several options and parameters to try for our approach, we have done a wide range of experiments and trials. In this section, we are going to define and explain a concrete experiment which is useful to understand behaviours and make comparisons. However, before introducing it we explain the main conclusions after using and tuning the approach parameters.

- When computing the Laplacian with **networkx**, we observe a trade-off between duration and performance since by using the unnormalized option it takes way more time but it leads to way better score function results when clustering. The reason why we are getting so much better scores is that the unnormalized Laplacian serves in the approximation of the minimization of our objective function as it is shown in [7].
- We obtain better results without eigenvectors normalization but without big differences comparing to normalizing by embedding (rows). However, we get very bad results when normalizing by eigenvectors (columns).
- When trying to use only the second smallest eigenvector we get very decent results, very competitive when comparing to keeping more eigenvectors.
- When inverting the Laplacian matrix as mentioned before, we do not improve performance.
- The number of eigenvectors kept is very important and we have played with selecting less and more than k, which leads to the top 10 results in the competition.
- By computing the Laplacian matrix manually instead of using the module **networkx** we do not observe differences.
- By iterating through different seeds we get different results for the same configurations and clustering methods, that is why we played with this parameter too in the competition.

Now, we explain and give numerical conclusions of the main experiment. By fixing some parameters we have tried all implemented clustering methods and in the case of the customized KMeans with **pyclustering** we test three different distance metrics as well as different centroids initializations randomly or with KMeans++. The rest fixed parameters are: number of unnormalized eigenvectors, unnormalized Laplacian with **networkx**, and seed 0. Due to the time restrictions and the fact that the biggest graphs are really time consuming, we have developed

the experiment on the two smaller graphs.

On the other hand, we have set a baseline output for each graph which corresponds to a random cluster assignation of each node. This baseline was useful in the beginning of our experiments in order to see whether we were improving a trivial solution. The score function results achieved by it are:

- ca-Qrc: 6
- Oregon-1: 15
- soc-Epinions1: 90
- web-Notredame: 120
- roadNet-CA: 100

In the following tables, you can find the experiment results. Note that d1, d2, d3 refer to Minkowski, Chebyshev, Euclidean and c1, c2 refer to random and KMeans++ centroids initialization and the time and memory usage stated on the table is without taking in account the computation of the eigenvectors of the laplacian since this is only calculated once, saved as a pickle file and used by all the methods.

Method	**Score**	**Memory**	**Time**
KMeans **pyclustering** d1 c1	0.118	139.844 MiB	1.243 seconds
KMeans **pyclustering** d1 c2	0.154	140.078 MiB	1.778 seconds
KMeans **pyclustering** d2 c1	0.200	139.941 MiB	1.258 seconds
KMeans **pyclustering** d2 c2	0.159	140.188 MiB	1.861 seconds
KMeans **pyclustering** d3 c1	0.200	139.871 MiB	1.534 seconds
KMeans **pyclustering** d3 c2	0.084	139.914 MiB	1.196 seconds
KMeans **scipy**	0.172	134.125 MiB	1.644 seconds
KMeans **sklearn**	0.063	136.433 MiB	0.505 seconds
XMeans **pyclustering**	0.084	139.652 MiB	1.724 seconds
Agglomerative **pyclustering**	0.125	139.387 MiB	275.884 seconds

Table 2: Experiments on graph *ca-GrQc* taking 10 eigenvectors as a high-dimensional embedding.

Method	**Score**	**Memory**	**Time**
KMeans **pyclustering** d1 c1	0.990	157.027 MiB	0.465 seconds
KMeans **pyclustering** d1 c2	1.361	158.105 MiB	0.502 seconds
KMeans **pyclustering** d2 c1	1.631	157.301 MiB	0.435 seconds
KMeans **pyclustering** d2 c2	1.818	158.180 MiB	0.477 seconds
KMeans **pyclustering** d3 c1	0.867	158.203 MiB	0.434 seconds
KMeans **pyclustering** d3 c2	1.217	157.934 MiB	0.477 seconds
KMeans **scipy**	1.449	142.039 MiB	0.978 seconds
KMeans **sklearn**	0.654	145.930 MiB	1.059 seconds
XMeans **pyclustering**	0.867	156.879 MiB	0.484 seconds

Table 3: Experiments on graph *Oregon-1* taking 13 eigenvectors as a high-dimensional embedding.

From table [2] we can see that the Agglomerative Hierarchical clustering takes much more time than the other approaches and does not compute a better partition, hence we decided not to push forward with this method for the other graphs. We can also observe in tables [2], [3] how sklearn

is the method that gives us the best score. We believe that the reason of having different scores between different KMeans methods could be the way some parameters are implemented in every library such as tolerance or number of maximum iterations. Moreover, in terms of memory usage, we do not observe a huge difference between methods (all of them are using sparse matrices) even though they come from different libraries.

The next tables were calculated only using the sklearn method and varying the number of smallest eigenvectors we keep as a high-dimensional embedding of the graph. Here we can see how much this value changes the outcome of score.

# Smallest Eigenvectors	**Score**
1	6.5424
2-8	0.0836
9-10	0.0627

Table 4: Experiments on graph *ca-GrQc* on how taking different number of eigenvectors as a high-dimensional embedding changes outcome.

# Smallest Eigenvectors	**Score**
3	1.110
6	0.7220
9	0.6639
12-18	0.6341

Table 5: Experiments on graph *Oregon-1* on how taking different number of eigenvectors as a high-dimensional embedding changes outcome.

The experiments were run in a server called *brute.aalto.fi* provided by Aalto University with the following characteristics:

- 32 Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz x86_64 Architecture
- 256 GiB System memory

5 Competition results

In this section, we will explain our optimal configuration and output by graphs which correspond to our final competition submission. We have tried several configurations and combinations of parameters (explained in previous sections), mainly including different numbers of eigenvectors kept for the clustering algorithms and different seeds, which has been the key to get top 10 positions in the competition.

We have used the following same parameters for all of them: clustering method KMeans **sklearn** which as mentioned before uses Euclidean distance and KMeans++ centroids initialization, computation of Laplacian matrix with **networkx** without normalization and without inverting, neither eigenvectors normalization. Moreover, in the following table you can find the number of smallest eigenvectors kept and random seed for each graph, as well as the score and final position in the competition.

GraphID	Eigenvectors	Seed	Score	Position
ca-Qrc	10	0	0.0627414293	1
Oregon-1	13	1	0.6003758692	1
soc-Epinions1	27	8	0.5503901003	8
web-Notredame	60	0	0.0306827622	12
roadNet-CA	100	0	0.2626139803	14

6 Conclusions

Overall, we can conclude that we have implemented successfully different methods to partition large networks and we have experienced empirically how useful can be to use the properties found in the spectra of a graph. We also have learnt the hard way how difficult can be to work with large networks in terms of time and memory usage; in fact, some of our first approaches which involved Hill Climbing and Simulated Annealing methods were too harsh to implement in this kind of networks.

Furthermore, as it can be seen in the section [4], we have also observed how clustering on a higher dimensional space than the one stated in the original algorithm [3], that is to say taking more eigenvectors than k, gave us better results. We hypothesize that this is because the additional eigenvectors provide more additional information of the structure of the graph.

References

[1] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, pp. 7821–7826, June 2002.

[2] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," 2004.

[3] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," *Neural computation*, vol. 15, no. 6, pp. 1373–1396, 2003.

[4] U. von Luxburg, "A tutorial on spectral clustering," 2007.

[5] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, pp. 849–856, 2002.

[6] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 888–905, 2000.

[7] J. B. Buger, "Why laplacian matrix need normalization and how come the sqrt of degree matrix?," Jan 2015.